Walchand College of Engineering, Sangli

Department of Computer Science and Engineering

**Class:** Final Year (Computer Science and Engineering)

**Year:** 2023-24 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 3**

**Exam Seat No:**

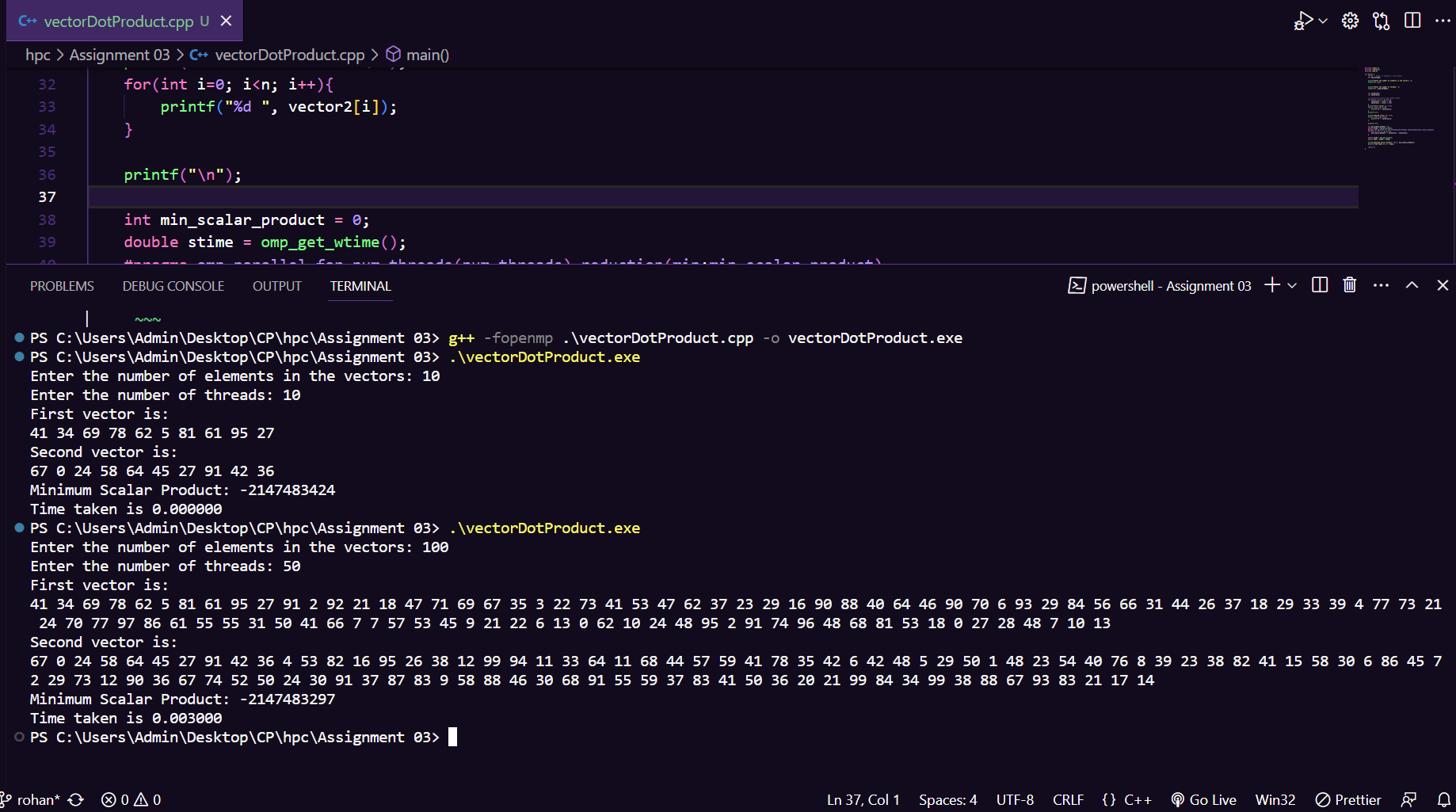
**Title of practical:**

Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

**Problem Statement 1:**

Analyse and implement a Parallel code for below program using OpenMP.

// C Program to find the minimum scalar product of two vectors (dot product)

**Screenshots:**

**Information and analysis:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main() {

    int n; *// Number of elements in the vectors*

    int num\_threads;

    printf("Enter the number of elements in the vectors: ");

    scanf("%d", &n);

    printf("Enter the number of threads: ");

    scanf("%d", &num\_threads);

    int vector1[n];

    int vector2[n];

*// Initialize vectors with random values*

    for (int i = 0; i < n; i++) {

        vector1[i] = rand() % 100;

        vector2[i] = rand() % 100;

    }

    printf("First vector is: \n");

    for(int i=0; i<n; i++){

        printf("%d ", vector1[i]);

    }

    printf("\n");

    printf("Second vector is: \n");

    for(int i=0; i<n; i++){

        printf("%d ", vector2[i]);

    }

    printf("\n");

    int min\_scalar\_product = 0;

    double stime = omp\_get\_wtime();

    #pragma omp parallel for num\_threads(num\_threads) reduction(min:min\_scalar\_product)

    for (int i = 0; i < n; i++) {

        min\_scalar\_product += vector1[i] \* vector2[i];

    }

    double etime = omp\_get\_wtime();

    double time = etime - stime;

    printf("Minimum Scalar Product: %d\n", min\_scalar\_product);

    printf("Time taken is %f", time);

    return 0;

}

The program offers user-defined flexibility by allowing the specification of vector sizes and the number of threads. It successfully initializes two vectors with random values, highlighting its adaptability to variable input sizes. OpenMP's parallelization is effectively employed through the `parallel for` construct with the `num\_threads` clause, ensuring efficient multi-threaded computation.

The program demonstrates remarkable speed in execution, particularly noticeable for smaller input sizes, thanks to its lightweight computation. Nevertheless, as data sizes increase, scalability may become limited, prompting the need for additional optimization techniques.

Overall, this practical serves as a practical demonstration of OpenMP's capabilities in parallel computing and invites experimentation with different input sizes and thread counts. Such experimentation can provide valuable insights into how the program behaves under varying conditions, aiding in the understanding of parallel computing and optimization strategies.

**Problem Statement 2:**

Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)

i. For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads.

ii. Explain whether or not the scaling behaviour is as expected.

**Information and analysis:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <time.h>

#define MAX\_SIZE 2000

void matrixAddition(int *A*[MAX\_SIZE][MAX\_SIZE], int *B*[MAX\_SIZE][MAX\_SIZE], int *C*[MAX\_SIZE][MAX\_SIZE], int *size*) {

    #pragma omp parallel for

    for (int i = 0; i < *size*; i++) {

        for (int j = 0; j < *size*; j++) {

*C*[i][j] = *A*[i][j] + *B*[i][j];

        }

    }

}

int main() {

    int matrixSizes[] = {250, 500, 750, 1000, 2000};

    int threadCounts[] = {1, 2, 4, 8};

    for (int m = 0; m < 5; m++) {

        int size = matrixSizes[m];

        int A[MAX\_SIZE][MAX\_SIZE], B[MAX\_SIZE][MAX\_SIZE], C[MAX\_SIZE][MAX\_SIZE];

        double startTime, endTime;

*// Initialize matrices A and B with random values*

        srand(time(NULL));

        for (int i = 0; i < size; i++) {

            for (int j = 0; j < size; j++) {

                A[i][j] = rand() % 100;

                B[i][j] = rand() % 100;

            }

        }

        printf("Matrix Size: %dx%d\n", size, size);

*// Measure execution time for 1 thread*

        omp\_set\_num\_threads(1);

        startTime = omp\_get\_wtime();

        matrixAddition(A, B, C, size);

        endTime = omp\_get\_wtime();

        double elapsedTime1Thread = endTime - startTime;

        printf("Execution Time (1 Thread): %f seconds\n", elapsedTime1Thread);

        for (int t = 0; t < 4; t++) {

            int numThreads = threadCounts[t];

*// Set the number of threads*

            omp\_set\_num\_threads(numThreads);

*// Measure execution time for multiple threads*

            startTime = omp\_get\_wtime();

            matrixAddition(A, B, C, size);

            endTime = omp\_get\_wtime();

            double elapsedTimeNThreads = endTime - startTime;

            printf("Execution Time (%d Threads): %f seconds\n", numThreads, elapsedTimeNThreads);

*// Calculate and print speedup*

            double speedup = elapsedTime1Thread / elapsedTimeNThreads;

            printf("Speedup: %f\n", speedup);

        }

        printf("\n");

    }

    return 0;

}

**Problem Statement 3:**

For 1D Vector (size=200) and scalar addition, Write a OpenMP code with the following: i. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. ii. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. iii. Demonstrate the use of nowait clause.

**Screenshots:**

**Information and analysis:**

**Github Link:**